AD-A250 438

TION PAGE

Form Approved
OMB No. 0704-0188

F
n
ir
v

ɘ 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and information. Send comments regarding this burden estimate or any other aspect of this collection of information, ervices, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, duction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE April 1992	3. REPORT TYPE AND DATES COVERED Special Technical

4. TITLE AND SUBTITLE Using Consistent Subcuts for Detecting Stable Properties	5. FUNDING NUMBERS NAG 2-593
6. AUTHOR(S) Keith Marzullo and Laura Sabel	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Keith Marzullo, Assistant Professor Department of Computer Science Cornell University	8. PERFORMING ORGANIZATION REPORT NUMBER 92-1277
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) ARPA/ISTO	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Please see page 1.

DTIC
ELECTE
MAY 12 1992
S B D

14. SUBJECT TERMS	15. NUMBER OF PAGES 27
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UNLIMITED

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

Using Consistent Subcuts for Detecting Stable Properties*

Keith Marzullo
Laura Sabel

TR 92-1277
(replaces TR 91-1205)
April 1992

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This work was supported by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593 and by grants from IBM and Siemens. The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy or decision.

Using Consistent Subcuts for Detecting Stable Properties*

Keith Marzullo Laura Sabel

Cornell University
Department of Computer Science

23 March 1992

Abstract

We present a general protocol for detecting whether a property holds in a distributed system, where the property is a member of a subclass of stable properties we call the *locally stable properties*. Our protocol is based on a decentralized method for constructing a maximal subset of the local states that are mutually consistent, which in turn is based on a weakened version of vector time stamps. The structure of our protocol lends itself to refinement, and we demonstrate its utility by deriving some specialized property-detection protocols, including two previously-known protocols that are known to be efficient.

1 Introduction

It is conceptually simple to determine whether the global state of a distributed system satisfies a stable property; that is, a property Φ that satisfies $\Phi \Rightarrow \Box\Phi$. One can have a process use a snapshot algorithm such as the one given in [CL85] to collect the relevant local and channel states and then test to see if the condition holds over the collected state. This technique can

*This work was supported by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593, and by grants from IBM and Siemens. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

be used to detect any stable property. However, for many stable properties of interest, such as deadlock, termination, and lack of a token, there exist specialized protocols (for example, [Mis83,Mat87,BT84,CMH83]) that are more efficient than a straightforward application of [CL85]. As well as being more efficient, many of these specialized protocols are very elegant and their relation to snapshots is not apparent.

It would be useful if one could derive such special-purpose protocols by refinement of a general snapshot protocol. Unfortunately, the protocol of [CL85] was not developed with refinement in mind, and we have not found it conducive to such refinement. In this paper, we present a different protocol for detecting stable properties that has proven to be more conducive to refinement.

A naive general detection protocol is as follows: every time a process executes an event, it appends its current state to a queue maintained in local memory. A separate process p_0 periodically retrieves these queues of local histories and extracts from them the latest global state. Process p_0 then tests to see if the property holds in this global state. Unfortunately, this protocol is impractical since it has a large execution overhead and requires unbounded local memory. This can be fixed (at a cost of generality) by having each process record only its current state at appropriate times and by having p_0 consider some subset of these local states that could be part of a sensible global state of the system. Not all stable properties can be detected this way, but it turns out that most stable properties that have been discussed in the literature can.

In this paper, we present a method to detect a subclass of stable properties. The method can be easily expressed as a decentralized protocol and can be customized for different properties in order to yield efficient special-purpose protocols. We demonstrate its utility by using it to derive such protocols including two previously-known protocols that are known to be efficient.

2 Definitions

2.1 System Model

We consider an asynchronous distributed system consisting of a set of n nonfaulty processes $P = \{p_1, p_2, \ldots, p_n\}$. Between any two processes p_i and p_j there exist two unidirectional nonfaulty FIFO channels: $C_{i,j}$ from p_i to p_j and $C_{j,i}$ from p_j to p_i. These channels have unbounded delivery time, and processes communicate only by sending and receiving messages over these channels.

Processes execute *events*, which are partitioned into *send* events, *receive* events, and *local* events. We will denote the tth event executed by process p_i as e_i^t and the resulting local state σ_i^t. Thus, the execution of process p_i can be denoted $(\sigma_i^0\ e_i^1\ \sigma_i^1\ e_i^2\ \sigma_i^2\ \cdots)$. Note that the state σ_i^t reflects the execution of events e_i^1 through e_i^t. When the ordinality of an event or state is not important, we will drop the superscript, e.g. the execution of event e_i results in the local state σ_i.

An arbitrary collection of local states may not constitute a sensible global state: the local state of a process in the collection may reflect the receipt of a message while no process' local state reflects the sending of that message. Such sets of local states are called *inconsistent*; a sensible collection of local states is called *consistent*. ([CL85])

A *global state* is defined to be a consistent set $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ of the processes' local states. We assume that channel states are captured in the local states of the processes. There are many ways to do this, for example by having each process maintain a history of all messages that it sends and receives. In practice, one must ensure that the representation of the channel states does not require an unbounded number of messages to be recorded.

A *consistent cut* is defined to be a set of events $C = \{e_1, e_2, \ldots, e_n\}$ such that the set of states $\{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ produced by C is a global state. Thus, each consistent cut has a corresponding global state, and vice versa. In this sense, global states and consistent cuts are equivalent notions. When defining properties of a distributed system, it is convenient to refer to states; our protocol uses events. For this reason, we define both global states and consistent cuts for use in different contexts.

A *property* is a predicate over the global state of the system. A *stable* property is an invariant: once it becomes true, it continues to be true[1]. The most commonly studied examples of stable properties in distributed systems are deadlock of a subset of the processes, termination of a distributed computation, and the lack of a token among the processes. There are, of course, other stable properties of interest. For example, in a token-passing system that can lose but not generate tokens, the predicate "there are no more than two tokens in the system" is a stable property.

Let $\sigma_i|\Phi$ (read σ_i *relative to* Φ) be the values of the subset of variables of σ_i that are referenced in the formulation of property Φ. An event e_i^t is *relevant* to a property Φ if $\sigma_i^{t-1}|\Phi \neq \sigma_i^t|\Phi$; that is, if e_i^t changes p_i's local state relative to Φ by changing the value of a variable in the formulation of Φ. For example, if Φ is "a subset of the processes are deadlocked" then the relevant events are those that request a resource and those that grant a resource, since Φ is formulated in terms of resource requests and grants. Note that local events, send events, and/or receive events can be relevant, depending on how Φ is formulated.

2.2 Vector Clocks

Our protocol is based on a variant of *vector clocks* [Mat89]. In the usual definition of a vector clock V, each event e_i has an n-component vector $V(e_i)[1..n]$ associated with it. $V(e_i)$ is called the *vector timestamp* of e_i. The components of $V(e_i)$ are:

- $V(e_i^t)[i] = t$; that is, $V(e_i^t)[i]$ is the number of events that p_i has executed up to and including e_i^t.

[1]Some authors define an invariant property to be one that is valid in all states of the system.

- $V(e_i^t)[j], j \neq i$ is the number of events p_j has executed that causally precede e_i^t.

As an example, Figure 1 shows a space–time diagram of a two-process system with the events labeled by vector clocks.

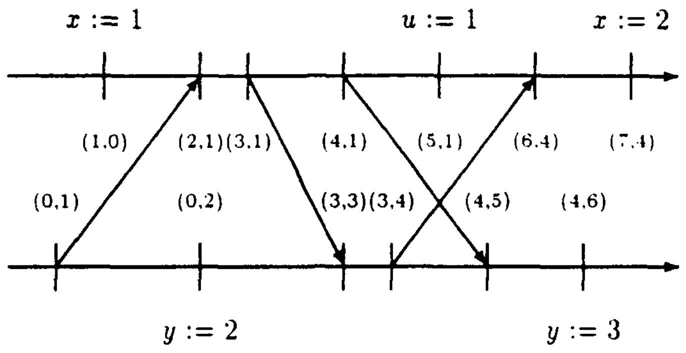

Figure 1: Execution with vector clocks.

A simple implementation of vector clocks has process p_i maintain an n–element vector V_i of counters. Process p_i increments $V_i[i]$ whenever it executes an event e_i If e_i is a local event or a send event, then $V(e_i) = V_i$. If e_i is a send event, then p_i includes V_i in the message. If e_j is the corresponding receive event, then process p_j sets $\forall k : k \neq j : V_j[k]$ to the maximum of the previous value of $V_j[k]$ and the value of $V_i[k]$ in the message, and $V(e_j) = V_j$.

The following three relations hold between vector clocks and global states, where $\rightarrow$ is the *happened-before* relation defined in [Lam78]. Equation 1 defines the happened-before relation in terms of vector clocks. Equation 2 defines when two events are consistent with each other (we call two such events *pairwise consistent*), and Equation 3 defines when a set of local states $\{\sigma_1, \ldots, \sigma_n\}$ produced by events $\{e_1, \ldots, e_n\}$ comprise a (consistent) global state:

$$\forall i, j : i \neq j : V(e_i)[i] \leq V(e_j)[i] \quad \equiv \quad e_i \rightarrow e_j \tag{1}$$

$$\begin{aligned}(V(e_i)[i] \geq V(e_j)[i]) \wedge & \\ (V(e_j)[j] \geq V(e_i)[j]) &\equiv e_i \text{ and } e_j \text{ are pairwise consistent} \quad (2) \\ \forall i,j\colon V(e_i)[i] \geq V(e_j)[i] &\equiv \{\sigma_1, \ldots, \sigma_n\} \text{ is a global state} \quad (3)\end{aligned}$$

Equation 2 can be derived by noting that two events e_i^t and e_j^u are inconsistent only if (without loss of generality) $e_i^t \rightarrow e_j^u \wedge \exists e_i^{t'} : e_i^t \rightarrow e_i^{t'} \rightarrow e_j^u$, or in terms of vector clocks, $V(e_i^t)[i] < V(e_j^u)[i]$. Equation 3 can be derived from Equation 2 by noting that all events in a consistent cut are pairwise consistent. Observe that $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ is a global state if and only if $C = \{e_1, \ldots, e_n\}$ is a consistent cut.

When vector clocks are used in actual protocols, not all events cause a process' vector clock to be updated. For example, some causal broadcast protocols are based on vector clocks that are updated only at send or broadcast events; other events (i.e., receive and local events) do not increment the local component of V [Pet87,BSS90]. For our purposes, only the execution of relevant events update the local counter of a vector clock. This is because the execution of a nonrelevant event does not change the state of the system with respect to Φ.

If not all send events are relevant events, however, then Equations 2 and 3 need not hold. For example, suppose e_i^t and e_j^u are relevant events that are pairwise inconsistent: $e_i^t \rightarrow e_j^u \wedge \exists e_i^{t'} : e_i^t \rightarrow e_i^{t'} \rightarrow e_j^u$. If no such $e_i^{t'}$ is a relevant event, then $V(e_i^t)[i] = V(e_j^u)[i]$ and $V(e_j^u)[j] > V(e_i^t)[j]$, which satisfies the left side of Equation 2 even though e_i^t and e_j^u are pairwise inconsistent. On the other hand, $\sigma_i^t|\Phi = \sigma_i^{t'}|\Phi$, and so the fact that e_i^t and e_j^u are pairwise inconsistent is irrelevant with respect to Φ, as long as $e_i^{t'}$ and e_j^u are pairwise consistent.

We therefore define a type of vector clock for which a weaker version of Equation 2 holds. Let the *weak vector clock V_Φ for Φ* be the vector clock in which $V_\Phi(e_i)[i]$ counts only the events relevant to Φ that p_i has executed through e_i. Therefore, the vector timestamp associated with several events of p_i may have the same value, but all such events result in the same local state relative to Φ.[2] For example, in the case of deadlock the relevant events are sending a request for a resource, sending a grant of a resource, and receiving a grant of a resource (see Section 5.2). If a process p requests a resource and then sends an unrelated message, then the send event does not change the local state of p with respect to possible deadlock. So, the send event is given the same weak vector timestamp as the resource request event.

We say that two events e_i^t and $e_i^{t'}$ are *equivalent with respect to* Φ, written $e_i^t \sim_\Phi e_i^{t'}$, if e_i^t and $e_i^{t'}$ have the same weak vector timestamp. We say that two local states σ_i^t and $\sigma_i^{t'}$ are equivalent with respect to Φ, written $\sigma_i^t \sim_\Phi \sigma_i^{t'}$, if $e_i^t \sim_\Phi e_i^{t'}$. Similarly, two global states $\Sigma = \{\sigma_1^{t_1}, \ldots, \sigma_n^{t_n}\}$ and $\Sigma' = \{\sigma_1^{t'_1}, \ldots, \sigma_n^{t'_n}\}$ are equivalent with respect to Φ, written $\Sigma \sim_\Phi \Sigma'$, if for all i, $\sigma_i^{t_i} \sim_\Phi \sigma_i^{t'_i}$. The following versions of Equations 1, 2, and 3 hold for both vector clocks and weak vector clocks:

$$\begin{aligned}
\forall i,j: i \neq j: V_\Phi(e_i^t)[i] \leq V_\Phi(e_j^u)[i] \quad &\equiv \quad \exists e_i^{t'}, e_j^{u'} : (e_i^{t'} \sim_\Phi e_i^t) \wedge (e_j^{u'} \sim_\Phi e_j^u): \\
&\qquad e_i^{t'} \rightarrow e_j^{u'} \qquad\qquad (4) \\
(V_\Phi(e_i^t)[i] \geq V_\Phi(e_j^u)[i]) \wedge \\
(V_\Phi(e_j^u)[j] \geq V_\Phi(e_i^t)[j]) \quad &\equiv \quad \exists e_i^{t'}, e_j^{u'} : (e_i^{t'} \sim_\Phi e_i^t) \wedge (e_j^{u'} \sim_\Phi e_j^u): \\
&\quad e_i^{t'} \text{ and } e_j^{u'} \text{ are pairwise consistent} \quad (5) \\
\forall i,j: V_\Phi(e_i)[i] \geq V_\Phi(e_j)[i] \quad &\equiv \quad \exists \text{ global state } \Sigma': \\
&\quad \Sigma' \sim_\Phi \{\sigma_1, \ldots, \sigma_n\} \qquad\qquad (6)
\end{aligned}$$

Figure 2 shows weak vector clock values for the execution shown in Figure 1, where we assume that the predicate of interest references x and y, but not u nor any of the channel states. Note that although the events $x := 1$ and $y := 3$ do not form a consistent cut, their timestamps satisfy Equation 6 since there exist several cuts equivalent to this inconsistent cut (all necessarily having $\langle x = 1, y = 3 \rangle$) and they are therefore consistent with

[2] Note that two events of different processes may have the same weak vector timestamp as well.

respect to Φ.

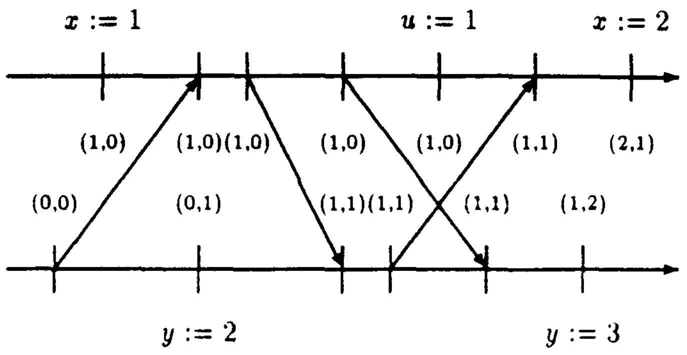

Figure 2: Execution with weak vector clocks.

2.3 Locally Stable Properties

Our protocol will detect a subset of the stable properties that we call *locally stable* properties. Informally, a stable property Φ is locally stable if no process involved in the property can change its state relative to Φ an unbounded number of times once Φ holds. For example, suppose Φ is "processes p_i and p_j are deadlocked." The property Φ is locally stable because once Φ becomes true, neither p_i nor p_j, the processes involved in Φ, can execute any event that could affect Φ (e.g., requesting or granting a resource). Hence, $\sigma_i|\Phi$ and $\sigma_j|\Phi$ remain constant once Φ holds.

More formally, let $\mathcal{G}$ be the set of all global states that the system can attain. For any $\Sigma \in \mathcal{G}$, define $\Sigma|\Phi$ to be the subset of Σ that is referenced in the formulation of Φ, and given a set of processes A define Σ_A to be the subset of Σ that consists of the states of the processes in A. We will call Φ locally stable if it is stable and if it satisfies the following condition: consider any $\Sigma \in \mathcal{G}$ that satisfies Φ, and let A be the set of processes p_i such that $\sigma_i|\Phi$ does not change an unbounded number of times in any execution starting at Σ. Then, for all $\Sigma' \in \mathcal{G}$ such that $\Sigma'_A|\Phi = \Sigma_A|\Phi$, Φ holds in Σ'.

In other words, Φ can be determined from only the states of the processes in A. Note that A can be empty, but only for trivial stable properties; if A is empty, then Φ can be determined without knowledge of the state of any process or channel and must therefore be valid or not valid in all states. For this reason, we will assume in this paper that A is nonempty.

The most commonly studied stable properties—deadlock, termination, and lack of a token—are all locally stable. For example, if Σ is a deadlock state, then A includes the deadlocked processes, and so the presence of deadlock can be determined by considering only the states of the processes in A. An example of a stable property that is not locally stable is the property "there are no more than $k : k > 0$ tokens" in a system where tokens cannot be created but can be lost when passed. This is because if Σ is a state in which there are k tokens, then every process can execute a relevant event an unbounded number of times (namely, it can pass tokens), thereby changing its local state relative to Φ an unbounded number of times, and so A is empty. The property, however, is not valid in all states of the token passing system.[3]

For most locally stable properties of interest, the processes in A cannot change their local states relative to Φ at all once Φ holds (i.e., the bound on the number of future relevant events that they can execute is zero). In this case, if our protocol presented below is initiated in a state in which Φ holds, it is guaranteed to detect Φ. For locally stable properties for which the bound is not zero, however, the protocol may not detect Φ if initiated in a state in which Φ holds. However, the system will eventually reach a state in which all processes in A will execute no further relevant events. If the protocol is initiated in such a state, it is guaranteed to detect Φ. Thus,

[3] A need not be empty for a stable property to be not locally stable. For example, suppose Φ is again "there are no more than $k : k > 0$ tokens" and the token passing system consists of red and green processors. Furthermore, only red processors can lose tokens and a green processor never passes a token to a red processor. In this system, A is the set of green processors (green processors never execute a relevant event), yet the validity of Φ depends on the states of both the green and red processors. Hence, Φ is not locally stable.

though the protocol eventually detects Φ in all cases, it detects Φ "sooner" when the bound is zero.

3 Protocol

3.1 Basic Protocol

We first assume that a process p_0 will determine whether the global state of the processes $P = \{p_1, \ldots, p_n\}$ satisfies a locally stable property Φ. Later, we will change this protocol so that any number of processes in P may concurrently assume the role of p_0.

Our protocol is based on the notion of a *consistent subcut*—a set of events whose timestamps satisfy Equation 6. (The state of a single process is trivially a consistent subcut.) Informally, the protocol works as follows. Whenever a process p_i executes a relevant event e_i, p_i records in a buffer B_i its local state relative to Φ and the vector time stamp $V_\Phi(e_i)$ as $B_i.\sigma$ and $B_i.V$, respectively. Process p_0 periodically collects the values of the buffers in any order, yielding a set $\mathcal{B} = \{B_1, B_2, \ldots, B_n\}$. Once p_0 has constructed this set, p_0 determines if there exists a maximal consistent subcut of $\mathcal{B}$ such that the states associated with the timestamps in the subcut satisfy Φ. If p_0 can find such a subcut, then Φ must currently hold. Note that p_0 need not examine all consistent subcuts; if $A' \subseteq A$ and Φ holds in $\Sigma_\Phi|A'$, then Φ will also hold in $\Sigma_\Phi|A$, so we need examine only the maximal consistent subcuts of $\mathcal{B}$. Of course, Φ may be of the form $\forall p_i : \Psi(p_i)$, in which case only a full consistent cut will satisfy Φ.

Unfortunately, the number of maximal subcuts of a set of n weak vector clocks is $\Omega(2^n)$. Fortunately, it is not necessary for p_0 to examine all of these subcuts. Suppose the set of buffer values contains B_i and B_j that are inconsistent: $B_i.V[i] < B_j.V[i]$. These two states violate Equation 6, and so both cannot be part of the same consistent subcut. However, $B_i.V[i] < B_j.V[i]$ implies that p_i executed a relevant event between the time that $B_i.\sigma$ was recorded and the time that $B_j.\sigma$ was recorded. Therefore, p_0 need

- Each process $p_i \in P$ records σ_i and $V_\Phi(e_i)$ in buffer B_i upon executing a relevant event e_i.
- Periodically, p_0 collects all of the buffers B_i and extracts from them the latest subcut $\{\sigma_i : \forall j : B_j.V[i] \leq B_i.V[i]\}$.
- p_0 detects Φ if Φ holds on the latest subcut.

Figure 3: Basic Protocol

not consider subcuts containing $B_i.\sigma$: if the system is in a state such that the processes involved in Φ will execute no more relevant events, then $B_i.\sigma$ cannot be necessary for the detection of Φ and so need not be considered. Otherwise, the system will eventually reach such a state. If $B_i.\sigma$ is involved in determining Φ, then B_i will be recorded such that $B_i.V[i] \geq B_j.V[i]$ for all j. Thus, given a set of buffered values $\mathcal{B}$ and the partial order $\forall B_i, B_j \in \mathcal{B} : B_i \succ B_j \overset{\text{def}}{=} B_i.V[j] > B_j.V[j]$, p_0 need only find the greatest elements of $\mathcal{B}$ with respect $\succ$, which can be done in $\Omega(n^2)$ time.[4] We call this subcut the *latest subcut* of $\mathcal{B}$. The latest subcut is clearly a maximal subcut, since all states that are not part of the latest subcut are inconsistent with some state in the latest subcut. This gives us the protocol shown in Figure 3.

The soundness of this protocol is straightforward. We now argue that the protocol is complete as well; that is, if Φ holds, then our protocol will detect Φ. Let Σ be the first global state in which Φ holds. Since Φ is locally stable, there is a nonempty set of processes $\mathcal{A}$ each of which executes a bounded number of relevant events after Σ; these processes will not change their states relative to Φ nor update their vector clocks an unbounded number of times

[4]The greatest elements of $\succ$ can be found by discarding any values B_j such that $\exists i : B_i.V[j] > B_j.V[j]$, which can be done in $O(n^2)$ time using a straightforward algorithm. And, if all values are incomparable then all the values are greatest elements of $\succ$. To determine that they are all incomparable takes n^2 comparisons, and so the problem is $\Omega(n^2)$.

in any run starting at Σ. Suppose that p_0 initiates the protocol in or after Σ (i.e., when Φ holds). Because the processes in A can each execute only a bounded number of relevant events after Σ, and because message delivery time is finite, there is some global state Σ' reachable in finite time from Σ after which the processes in A execute no more relevant events. Therefore p_0 will eventually collect the states $\Sigma'_\Phi|A$. From the definition of $\succ$, the state of a process p_i in A must be in any latest subcut constructed by p_0 because p_i will execute no more relevant events. Since Φ is stable, Σ' satisfies Φ, and since Φ is locally stable, Φ can be detected by examining $\Sigma'_A|\Phi$. Hence, p_0 will detect Φ.

3.2 Decentralization

In the above protocol, p_0's role is to collect the local states, determine the latest subcut, and check if Φ holds in this subcut. We can decentralize these steps by collecting the local states in a token.

Consider a token K that consists of n entries $\langle D_1, \ldots, D_n \rangle$ where each entry $D_i = (B_i.\sigma, B_i.V[i])$; that is, D_i will hold the state of p_i relevant to Φ and the local component of p_i's vector clock when it generated $B_i.\sigma$. Assume that there exists a special value $\perp$ for D_i indicating that the state is not in the token; all of the D_i in K are initially set to $\perp$.

To determine whether Φ holds, a process generates a token K, inserts its state and vector clock value into K, and passes the token to any other process. When a process p_j receives a token K, it takes the following steps:

1. Set D_j to $(B_j.\sigma, B_j.V[j])$.

2. For all non-$\perp$ values of D_k that are not in the latest subcut, set D_k to $\perp$.

3. Determine whether the state values in K satisfy Φ. If so, then the detection is made; otherwise, p_j forwards the token to a process p_k, chosen fairly, that has $D_k = \perp$. If there is no such process, then p_j

can either drop the token or, when p_j computes a new value of B_j, p_j can restart at Step 1 with this token.

Note that when process p_j executes rule 2, B_j must be part of the latest subcut; if it were not, then there would exist a recorded value of B_ℓ in D_ℓ such that $B_j.V[j] < B_\ell.V[j]$. This implies that p_ℓ knows of a relevant event executed by p_j that results in a state causally after the state recorded in B_j, which violates the definition of B_j. Thus, only the earlier values D_k need be tested with respect to B_j. From above, the value B_k in D_k can be discarded if $B_k.V[k] < B_j.V[k]$. The value $B_k.V[k]$ is stored in $D_k.V$, so K carries enough information for p_j to make this test.

The resulting protocol is summarized in Figure 4. Note that we have no *a priori* restriction on how many tokens there can be in the system at any time or on the order in which the token is passed, other than that it is passed in a fair manner. These decisions can be made when the protocol is applied to a particular problem.

If this protocol is initiated in a state in which Φ holds and after which no process executes a relevant event, then Φ will be detected with no more than n token passes. However, if processes do execute relevant events after the protocol is initiated, then the initial detection may not be successful and the protocol must be restarted. If the number of relevant events that can be executed after Φ holds is bounded by λ, then detection can take up to an additional λn token passes. For large λ, our protocol could perform worse than a snapshot protocol. In practice, however, we do not expect λ to be large.

4 Termination Detection

We now instantiate the general protocol given above to obtain a protocol that detects *termination* in a distributed system. There are many variations of this property; the earliest that we know of is due to Dijkstra and Scholten [DS80]. The following definition is the same as that given

- Each process $p_i \in P$ records σ_i and $V_\Phi(e_i)$ in buffer B_i upon executing a relevant event e_i.
- When p_i wants to detect Φ, p_i generates a token $\langle D_1 := \bot, \ldots, D_n := \bot \rangle$, sets D_i to $(B_i.\sigma, B_i.V[i])$, and forwards the token to any other process.

When p_j receives a token:

- p_j sets D_j to $(B_j.\sigma, B_j.V[j])$.
- For each $D_k \ : k \neq j, D_k \neq \bot$, p_j sets $D_k := \bot$ if $D_k.V = B_k.V[k] < B_j.V[k]$.
- p_j determines if Φ holds on the state values in the token. If not, p_j forwards token to any p_k such that $D_k = \bot$.

Figure 4: Decentralized Protocol

in [Mis83].

All processes are either *active* or *idle*. Only active processes can send messages. An active process may become idle at any time, and an idle process can become active only upon receipt of a message. The system is ***terminated*** when all processes in the system are idle and there are no messages in transit.

The local state of a process relative to termination consists of whether the process is active or idle and whether there is a message on an incoming channel. Therefore, the events that are relevant to termination are sending a message, receiving a message, becoming idle, and becoming active. Each process will update its (weak) vector clock upon executing any of these events. Note that for this problem, we do not need to keep track of the contents of the messages exchanged between processes; only the number of messages is important. To capture the channel states, we have each process keep track of how many messages it has sent and received on each adjacent channel. The combined information of all of the processes will then yield the number of messages in transit on each channel: if p_i has sent more messages to p_j than p_j has received from p_i, then there is at least one message on channel $C_{i,j}$. In this way, we can represent the relevant channel states without recording an unbounded number of messages.

We instantiate the general protocol given in Section 3.2 as follows.

Each process p_i maintains the following local state variables:

- $active_i$: Boolean = true if and only if p_i is active.
- $send_i[1..n]$: Integer array. $send_i[j]$ = the number of messages that p_i has sent to p_j. All are initially 0.
- $recv_i[1..n]$: Integer array. $recv_i[j]$ = the number of messages that p_i has received from p_j. All are initially 0.

When p_i sends a message to p_j, $send_i[j]$ is incremented. When p_i receives a message from p_j, $recv_i[j]$ is incremented. When p_i becomes active or idle, $active_i$ is set appropriately.

At some point, an idle process will start the detection protocol by circulating a token as described in Section 3.2. The termination condition can only be evaluated over a total global state (as opposed to a consistent proper subset of the process states), so a positive determination can be made only by the process p_f that is the last to add its state to the token.

Process p_f detects termination if and only if the following three conditions hold:

1. The timestamps in the token form a consistent cut;
2. All processes are idle: $\forall i : active_i$ = false;
3. There are no messages in transit: $\forall i, j : send_i[j] = recv_j[i]$.

The following theorem and corollary show that item 1 is redundant. The theorem assumes for simplicity that the buffered states are not collected in a token; the corollary removes this assumption.

Theorem 1 *Let $\mathcal{B} = \{B_i : i = 1, 2, \ldots n\}$ be a a set of buffered state values that were recorded in a system that* does not *collect states in a token D. If in $\mathcal{B}, \forall i, j$:* $\text{send}_i[j] = \text{recv}_j[i]$, *then the global state defined by $\mathcal{B}$ is consistent: $\forall i, j : B_i.V[i] \geq B_j.V[i]$.*

Proof: Suppose by way of contradiction that item 3 holds over $\mathcal{B}$ but the timestamps in $\mathcal{B}$ form an inconsistent cut: $\exists i, j : B_i.V[i] < B_j.V[i]$. $B_j.V[i]$ is advanced only when p_j receives a message, and events local to p_j affect only $B_j.V[j]$. Therefore, in order for $B_j.V[i]$ to advance beyond the recorded $B_i.V[i]$, there must have been a chain of messages between p_i and p_j between the time that B_i was collected and the time that B_j was collected. This implies that there is some k such that the recorded $send_i[k] < recv_k[i]$, contradicting the assumption that item 3 holds.

□

Corollary 2 *Let $\mathcal{B} = \{B_i : i = 1, 2, \ldots n\}$ be a a set of buffered state values that were collected in a token D. If in $D, \forall i, j$:* $\text{send}_i[j] = \text{recv}_j[i]$, *then the global state defined by D is consistent: $\forall i, j : D_i.V[i] \geq D_j.V[i]$.*

Proof: None of the events executed in collecting the buffered states into a token are relevant. Hence, collecting the states in this way has no effect on their consistency with respect to termination. The buffered states will therefore be consistent with respect to termination when items 2 and 3 hold.

□

Corollary 2 implies that the vector clocks need not be maintained. Furthermore, these checks can be done incrementally. For example, we can assign a total order to the processes and have the token passed along that total order. When process p_k receives the token, it tests to see if

$$\neg active_k \wedge (\forall \ell : 1 \leq \ell < k : (send_k[l] = recv_l[k]) \wedge (send_l[k] = recv_k[l])).$$

If this condition does not hold, then p_k can drop the token. If the condition holds and $k = n$, then termination is detected; otherwise, p_k fills in D_k and passes the token to p_{k+1}.

This yields the protocol given in [Mat87] as the *channel counting* protocol, which requires only n messages to detect termination once it holds, and which can be further refined into a protocol that is space-efficient. This is a good example of how our general protocol, which constructs consistent (sub)cuts explicitly, can be used to derive a much simpler protocol that constructs consistent cuts implicitly.

5 Deadlock Detection

5.1 k-out-of-m Deadlock

We now instantiate the general protocol given in Section 3 to obtain a protocol that detects *k-out-of-m deadlock* in a distributed system. This problem was first formulated and solved in [BT84]. In this formulation, a process can request k resources from a pool of m resources.

A process is either active or blocked, where an active process is one that is not waiting for any other process. Active processes may issue k-out-of-m requests in the following way. When an active process p_i requires k processes to carry out some request, it sends **request** messages to each of the m processes that can perform this action. Process p_i then becomes blocked, and waits until the action requested is carried out by at least k of the m processes. A process can not send any further requests while blocked, but a process can receive **request** messages while blocked.

Only active processes can carry out a requested action. If a process p_j receives a request while active, it will either become blocked or carry out p_i's requested action within finite time. In the latter case, p_j will send a **grant** message to p_i. When p_i receives k **grant** messages, it becomes active again. It then relinquishes the requests made to the rest of the processes to which it sent **request** messages by sending them **relinquish** messages. We assume that the recipient of a **relinquish** message acknowledges the message and that the sender of a **relinquish** message waits for all acknowledgements before sending another **request** message. By doing so, we guarantee that

p_i can discard any **grant** messages received after the first k are received.

The state of a process p_i relative to k-out-of-m deadlock consists of the number of grants needed for p_i to become active and the current set of processes that p_i is waiting for. We capture this state by having each process keep track of the processes on which it is blocked and the number of **grant** messages that it has sent and received on each adjacent channel.

We instantiate the general protocol given in Section 3.2 as follows. Each process p_i maintains the following local state variables:

- k_i: Integer = the number of **grant** messages required for p_i to become active (initially 0).
- $g\_send_i[1..n]$: Integer array. $g\_send_i[j]$ is the number of **grant** messages that p_i has sent to p_j (all are initially 0).
- $g\_recv_i[1..n]$: Integer array. $g\_recv_i[j]$ is the number of **grant** messages that p_i has received from p_j (all are initially 0).
- wf_i: Integer set. These are the processes that p_i is waiting for. When p_i sends a **request** message to p_j, $wf_i := wf_i \cup \{j\}$; when p_i receives a **grant** message from p_j or sends a **relinquish** message to p_j, $wf_i := wf_i - \{j\}$.

Deadlock is determined by constructing and reducing the system waits-for graph. This graph is constructed as follows:

- a *waits-for* edge is drawn from p_i to p_j if $wf_i \ni j \wedge (g\_send_j[i] = g\_recv_i[j]))$. That is, p_i is waiting for a resource from p_j and no grant message is in transit from p_j to p_i.
- the number of grants κ_i needed for p_i to be unblocked is $k_i - |\forall j : g\_send_j[i] - g\_recv_i[j]|$. That is, κ_i is the number of grants that p_i is waiting for less the number of grants in transit to p_i.

Deadlock is tested by reducing this graph as follows: if an edge points from p_i to p_j and p_j is active, then the edge can be erased and κ_i can be reduced by one; and if a process has $\kappa_i = 0$, then all of its outgoing edges can be erased. The system is deadlocked if and only if there are edges that cannot be removed by following these two rules.

In this system, the relevant events of p_i are those that change wf_i, k_i, g_send_i and g_recv_i. Hence, the relevant events are requesting a resource, sending a **grant** message, receiving a **grant** message and sending a **relinquish** message. We can now argue that k-out-of-m deadlock is locally stable: a deadlocked process can execute only a bounded number of relevant events (namely, it can receive up to $k_i - 1$ **grant** messages), and any valid global state that contains the local states of the deadlocked processes still yields an irreducible waits-for graph.

The deadlock detection protocol is as follows. When a process p_i wishes to test for deadlock, p_i generates a token, fills D_i with $((wf_i, g\_send_i, g\_recv_i, k_i), B_i.V[i])$, and forwards the token to some $p_j \neq p_i$. Upon receiving a token, a process p_j sets D_j to $((wf_j, g\_send_j, g\_recv_j, k_j), B_j.V[j])$ and discards all values D_k that are inconsistent with B_j by setting D_k to $\perp$. p_j then checks to see if deadlock holds on the remaining values by constructing the waits-for graph and reducing it. If deadlock does not hold, then p_j forwards the token to any process p_k such that $D_k = \perp$.

We can improve this protocol by choosing the process to which the token is passed more carefully. Since we would like to detect deadlock as quickly as possible, the forwarding process should choose a process that is likely to add information leading to the detection of a deadlock. A reasonable choice is a process p_j such that $D_j = \perp$ and such that p_j is in wf_i for some $D_i \neq \perp$.

The full protocol is presented in Figure 5. We assume that the process p_i that generates the token does so because it suspects that it is involved in a deadlock; that is, wf_i is not empty.

5.2 RPC Deadlock Detection

1-out-of-1 deadlock is a special case of k-out-of-m deadlock that lends itself to further optimization. This type of deadlock is called *RPC deadlock* be-

```
when $p_i$ receives token $\langle D_1, \ldots, D_n \rangle$:
  begin
  $D_i.\sigma := k_i, g\_send_i, g\_recv_i, wf_i$;
  $D_i.V := B_i.V[i]$;
  for all $D_j : D_j.V < B_i.V[j] : D_j := \perp$;
  if there exists $p_j : (D_j = \perp) \wedge (\exists p_i : j \in wf_i)$
  then forward token to one of these $p_j$
  else begin
    construct waits-for graph;
    reduce waits-for graph;
    if graph is not fully reduced then signal deadlock
    else drop the token
    end
  end
```

Figure 5: Protocol for Detecting k-out-of-m Deadlock

cause it can occur in a remote procedure call system, where making a remote procedure call is analogous to requesting a resource from a single processor. The waits-for graph is constructed as for k-out-of-m deadlock, except that $k_i = |wf_i|$ and thus need not be represented in the wait-for graph. Furthermore, **relinquish** messages are not needed and the waits-for graph is reducible if and only if it does not contain a cycle.

We can instantiate our protocol for detecting RPC deadlock as follows. As before, the relevant events are requesting a resource (here, making an RPC request), sending a **grant** message (here, sending the reply to the RPC request), and receiving a **grant** message. Any blocked process p_i can decide to detect deadlock by generating an empty token, inserting its buffered state into D_i, and passing the token to the (single) process in wf_i. When p_j receives a token from p_i, p_j will verify that p_i *waits-for* p_j and will pass the token on to the process blocking p_j. A process detects deadlock when it receives a token that contains a complete cycle. The resulting protocol is shown in

```
RPCDeadlock($p_j$): cobegin
    do forever when ($wf_j \neq \emptyset$) and (waited too long)
        create empty token $K$;
        $K.D_j := (g\_send_i[wf_j], B_j.V[j])$;
        pass $K$ to $wf_j$
|| do forever when receive token $K$ from $p_i$
        if $K.D_j = \perp$ then
            if $wf_j \neq \emptyset$ and $K.D_i.g\_send_i[j] = g\_recv_j[i]$
                and $\forall K.D_k$: $K.D_k \neq \emptyset$: $K.D_k.V[k] \geq B_j.V[k]$
            then
                $K.D_j := (g\_send_i[wf_j], B_j.V[j])$;
                pass $K$ to $wf_j$
            else skip /* drop token $K$ */
        else
            if $K.D_i.g\_send_i[j] = g\_recv_j[i]$ then detect deadlock
            else skip /* drop token $K$ */
coend
```

Figure 6: RPC Deadlock Protocol. Original

Figure 6. Note that if the waits-for graph contains a d-cycle, then the token need be passed only d times.

This protocol can be further simplified by applying the following two theorems.

Theorem 3 *If* $(K.D_i.g\_send_i[j] = g\_recv_j[i])$ *then* p_i *has executed no relevant event since setting* $K.D_i$.

Proof: Assume $(K.D_i.g\_send_i[j] = g\_recv_j[i])$. The last relevant event that p_i can have executed before setting $K.D_i$ was to send a **request** message to p_j. The first relevant event that p_i can have executed after setting $K.D_i$ is the receipt of a **grant** message from p_j. Since $(K.D_i.g\_send_i[j] = g\_recv_j[i])$, p_j has sent no **grant** messages to p_i since p_i sent the request to p_j. Hence, p_i can have executed no relevant event since setting $K.D_i$.

□

Theorem 4 *If* $(K.D_i.g\_send_i[j] = g\_recv_j[i])$ *then no process that has set its value in* K *has subsequently executed a relevant event.*

Proof: Let ℓ be the number of values $D_k : D_k \neq \perp$, and assume that $(K.D_i.g\_send_i[j] = g\_recv_j[i])$. We will use induction on ℓ.

Base case $(\ell = 1)$. Follows directly from Theorem 3.

Induction case $(\ell > 1)$. By the induction hypothesis, no process prior to p_i had executed a relevant event when p_i received K. No process prior to p_i can execute a relevant event until p_i does, and by Theorem 3 p_i has not executed a relevant event since forwarding K to p_j.

□

Theorem 4 implies

$$(g\_send_i[j] = g\_recv_j[i]) \Rightarrow (\forall k : D_k \neq \emptyset : D_k.V = B_k.V[k] \geq B_j.V[k]).$$

Thus, the vector clocks can be omitted and the token need only carry the identity of the process that initiated the test for deadlock. The resulting protocol, shown in Figure 7, first appeared in [CMH83] specialized for $m = 1$.

```
RPCDeadlock($p_j$): cobegin
   do forever when ($wf_j \neq \emptyset$) and (waited too long)
      send ($g\_send_j[wf_j], j$) to $wf_j$
|| do forever when receive ($s, k$) from $p_i$
      if $k \neq j$ then
         if $wf_j \neq \emptyset$ and $s = g\_recv_j[i]$
         then send ($g\_send_j[wf_j], k$) to $wf_j$
      else
         if $s = g\_recv_j[i]$ then detect deadlock
coend
```

Figure 7: RPC Deadlock Protocol. Refined

The protocol in Figure 6 can be easily generalized to detect detect **and**-deadlock (m-out-of-m requests), since a cycle in the waits-for graph is equivalent to deadlock in this case as well. The only change necessary is that when p_j passes the token, it must replicate the token and pass a copy to each process in wf_j. With **and**-deadlock, however, a process can execute a relevant event while deadlocked—a deadlocked process can receive a proper subset of the required **grant** messages. Thus, if the waits-for graph contains a d-cycle, then even if tokens are generated by a deadlocked process and passed along cycles, such tokens may be dropped up to $n - d$ times before the deadlock is detected. However, the protocol in Figure 7 can be effectively run in parallel by having p_j send ($g\_send_j[u], k$) to all the processes $u \in wf_j$ in which case a token passed along a cycle will not be dropped. The resulting deadlock detection protocol is the one presented in [CMH83].

6 Conclusion

This paper defined a proper subclass of the stable properties which we denote the ***locally stable properties***. This subclass is interesting in that a process that is "involved" in establishing the stable property is limited in what it can do, and will eventually cease changing its local state with respect to the stable property. Hence, in order to detect a locally stable property, a consistent cut need not be explicitly constructed—the relevant local states will form a consistent subcut implicitly. This leaves only the problem of detection.

In order to make this observation, we needed to define consistent cuts with respect to a global state predicate, and slightly extend the notion of vector clocks to accommodate our definition. We then gave a simple and decentralized protocol that detects when a locally stable property occurs in a distributed system. The protocol can be easily refined, which we illustrate by refining it to a known protocol for termination detection, a new protocol for k-out-of-m deadlock detection, and known protocols for m-out-of-m deadlock detection.

In the reductions to the two known protocols, the vector clocks proved redundant. This was because the processes involved in Φ could execute *no* relevant events once they established the condition of interest, and the detection algorithm also ensured that the channels carried no undelivered relevant messages. In both cases, the receipt of a relevant message was the only relevant event that a process involved in Φ could execute, and so an empty channel between two processes implied pairwise consistency of the recorded states of those two processes. This observation is similar to one of the steps in the refinement of a termination protocol given in [CM86], yet we have not been able to refine our protocol to their termination protocol.

The class of locally stable properties was defined in proving the protocol correct. We would like to determine what kinds of properties are locally stable. We know of two general classes: the locally stable properties of distributed garbage detection, termination, and global deadlock can all be expressed as detecting no token in a generalized token passing system, yet deadlock of a subset of the processes does not seem to be so expressible. We are interested in whether there are other classes of locally stable properties.

Not all interesting stable properties are locally stable, however. For example, the property "the number of tokens is less than $k > 0$" in a token passing system that can lose but not regenerate tokens is stable but not locally stable. We do not know if there are protocols that are more message-efficient than snapshot protocols for detecting such properties.

Our work was motivated by trying to derive message-efficient special-purpose detection protocols from a general detection protocol. We have only been partially successful. Our protocol is most efficient when no process can execute a relevant event after the condition of interest holds. Furthermore, in our derivation of the m-out-of-m deadlock detection protocol in Section 5.2, our protocol could generate $O(n)$ extra messages. Hence, we would like to better understand the notion of relevant events and weak vector clocks.

Acknowledgements We would like to thank Özalp Babaoğlu, Gil Neiger, Fred Schneider, and Sam Toueg for their contributions to the ideas in this paper. We would also like to thank Ken Birman, Bard Bloom, Navin Budhiraja, Tushar Chandra, and Mark Wood for their valuable comments on earlier drafts of this paper.

References

[BSS90] K. Birman, A. Schiper, and P. Stephenson. Fast causal multicast. Technical Report TR-90-1105, Cornell University, April 1990. Submitted for publication.

[BT84] Gabriel Bracha and Sam Toueg. A distributed algorithm for generalized deadlock detection. In *Proceedings of the Third Symposium on Principles of Distributed Computing*, pages 285–301. ACM SIGPLAN/SIGOPS, August 1984.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CM86] K. Mani Chandy and Jayadev Misra. An example of stepwise refinement of distributed programs: Quiescence detection. *ACM Transactions on Programming Languages and Systems*, 8(3):326–343, July 1986.

[CMH83] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.

[DS80] Edsger W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Mat87] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, 1987.

[Mat89] Friedemann Mattern. Time and global states of distributed systems. In Michel Cosnard et. al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. North-Holland, October 1989.

[Mis83] Jayadev Misra. Detecting termination of distributed computations using markers. In *Proceedings of the Second Symposium on Principles of Distributed Computing*, pages 290–294. ACM SIGPLAN/SIGOPS, August 1983.

[Pet87] Larry L. Peterson. Preserving context information in an IPC abstraction. In *Proceedings of the 6th Symposium on Reliability in Distributed Software and Database Systems*, pages 22–31, March 1987.